Automatic Full Functional Verification of Clients of User-Defined Abstract Data Types

Jason Kirschenbaum, Bruce Adcock, Derek Bronish, Bruce W. Weide

The Ohio State University, Columbus OH 43210, USA
`{kirschen,adcockb,bronish,weide}@cse.ohio-state.edu`
`http://www.cse.ohio-state.edu/rsrg`

Abstract. A scalable approach to addressing the verifying compiler grand challenge must handle the following four categories of programs that arise from modern programming language constructs for modern software engineering tasks: (1) code that uses only built-in data types (*e.g.,* integers, records, arrays, some uses of pointers) to provide new functionality; (2) code that represents new user-defined abstract data types (ADTs) by using only built-in data types; (3) code that uses existing user-defined ADTs to provide new functionality, but no new types; and (4) code that represents new user-defined ADTs by layering them on existing user-defined ADTs. Progress has been reported on verifying—sometimes automatically—the first two kinds of programs. In contrast, progress on applying automated reasoning to the latter two kinds of programs has been elusive. Yet this is a critical piece of the verified software puzzle because such layering enables both vertical scaling of software and modular verification thereof. The OSU RESOLVE verification system can automatically prove full functional correctness of imperative code in category (3), by using a combination of carefully designed imperative language features, a sound and relatively complete proof system with an associated verification-condition generator, and both off-the-shelf and special-purpose theorem-proving tools.

1 Introduction

The long-term vision guiding our work is that of a future in which no production software is considered properly engineered unless it has been fully specified and automatically verified as satisfying these specifications, a "grand challenge" [1]. Based on a collection of benchmarks [2] along any plausible path toward a verifying compiler, we can identify various categories of code (summarized in the abstract above) that such a tool must be able to reason about. Significant progress has been reported by a number of projects, including Spark [3], Spec#/Boogie [4], and Jahob [5, 6], in automatically verifying code in category (1), and to some extent in category (2). The contribution of this paper is its demonstration of progress for category (3).

Why has progress been slow for code in categories (3) and (4)? The most recent Jahob paper [6] helps illustrate the answer. It reports experience with automated verification of several Java classes, all in category (2). A critical difference in going from this kind of Java code, to the kind of code immediately encountered in categories (3) and (4), is the impact of aliasing [7]. In Jahob's verified category (2) Java code, all aliasing considered lies inside an encapsulation boundary, and is *intentional* and *controlled*. It is possible to model exactly what is going on with respect to such internal aliasing as it is used in the data structure representation in a class defining a new ADT. For code in categories (3) and (4), however, the potential for aliasing extends into the client program to be verified. Aliases can be unintentional, unanticipated, and external to previously verified classes that define the ADTs used by the client. Among other disasters, a client program might keep an alias to an object reference stored in some separately verified collection class, and it might change the value of the object through that alias—without the collection class knowing about it. Representation invariants of the collection class that involve object values rather than object references can thereby be broken, rendering any purported verification of the collection class unsound. For example, consider the BinarySearchTree class verified by Jahob. The entries in the tree are `int`s, not `object`s. This hamstringing of generality is absolutely essential for the Jahob verification to succeed. If the entries were object references and the search-tree ordering were defined on object values (as it must be for the class to be useful), then a client could hold an alias to an object reference in the tree and—without detection by the BinarySearchTree class—change that object's value through the alias in a way that falsifies the representation invariant that makes binary search trees work.

In short, the aliasing permitted by Java effectively prevents Jahob from soundly and modularly verifying Java code in categories (3) and (4). This is not Jahob's fault. It is instead the price to be paid for trying to verify code written in a language with unrestricted aliasing, *i.e.,* a language that was not intended to support modular verification. Such verification is not impossible in principle, but it is far more difficult to specify and model situations such as these without massively higher complexity [7] than one encounters in category (2) code. Scalability of automated verification depends on the ability to layer code arbitrarily deeply as in categories (3) and (4), *i.e.,* to handle code that is "above the ground floor" of using only built-in types. This is the entire point of "abstraction." So, while the projects mentioned above provide a great start toward addressing the verifying compiler grand challenge, it is folly to imagine that they have even addressed (let alone solved) the full problem.

2 The OSU RESOLVE Language and Proof System

2.1 Tool Architecture

RESOLVE [8] is a tool-supported component-based software discipline, designed to permit automated verification without sacrificing run-time efficiency of the resulting code. It consists of a formal mathematical language used for expressing program specifications, and an imperative programming language for developing implementations. RESOLVE includes syntactic slots in code for mathematical annotations, *e.g.,* loop invariants, and progress metrics for loops and recursive operations.

RESOLVE is promising as an approach to modular verification of code in categories (3) and (4) because of its approach to aliasing, the bane of modular automated verification: there is none. The language has value semantics. Reference-like behavior, where it is needed, is provided by software components that control and manage aliasing internally; it is not a client-exposed feature of the language, and hence does not pervade all specifications and proofs. Data movement in RESOLVE is achieved via swapping [9], not traditional assignment. This leads to a slightly different programming style, offering the efficiency of standard imperative languages while also supporting easier specification and modular automated verification of code in categories (3) and (4).

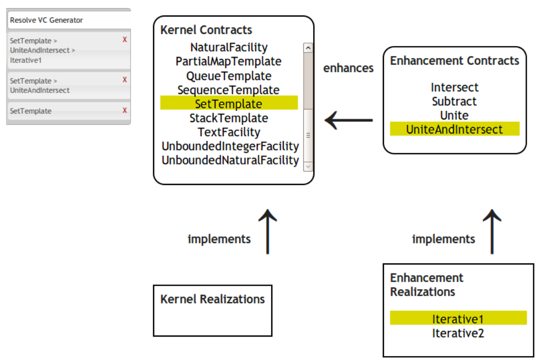


Fig. 1: Component Organization Screen

Given code purporting to implement a particular specification, the RESOLVE tool suite first generates verification conditions (VCs) — mathematical formulae whose truth corresponds to the correctness of the code. It then appeals to one or more back-end provers to establish the validity of these VCs. Finally, the results of the prover's work are reported to the user.

There are two groups working on qualitatively different RESOLVE tools—one at Clemson University, and ours at Ohio State University. This paper discusses the work of the latter.

The prototype verification system is demonstrated via a web interface. Fig. 1 shows a screen shot of the component selection process, in which the user chooses the code to verify. The list of known ADT interface contracts is shown in the "Kernel Contracts"

```
contract SetTemplate (type Item)

  uses UnboundedIntegerFacility

  math subtype SET_MODEL
    is finite set of Item

  type Set is modeled by SET_MODEL
    exemplar s
    initialization ensures
      s = empty_set

  procedure Add (updates s: Set,
                 clears x: Item)
    requires
      x is not in s
    ensures
      s = #s union {#x}

  procedure RemoveAny (updates s: Set,
                       replaces x: Item)
    requires
      s /= empty_set
    ensures
      x is in #s  and s = #s \ {x}

  function IsEmpty (restores s: Set)
                  : control
    ensures
      IsEmpty = (s = empty_set)
...

end SetTemplate
```

Fig. 2: Partial `SetTemplate` Contract

box, with the SetTemplate ADT selected. Once an ADT's contract has been selected, possible enhancements to it are populated in the "Enhancement Contracts" box. Since the UniteAndIntersect enhancement has been selected, the bottom-right box, "Enhancement Realizations," is populated with all its current purported implementations. In this box, the Iterative1 version has been selected.

Aside from demonstrating the appearance of the tool's interface, this figure also illustrates the highly component-oriented separation of concerns that the RESOLVE discipline encourages. The core functionality of a ADT is specified in its contract and implemented by a "Kernel Realization"; new features are layered on top via enhancements. Note that Iterative1 is exactly the kind of code that a client commonly writes in category (3): it takes a black-box view of existing user-defined ADTs to implement new functionality.

The web interface allows contract specifications to be viewed, as is shown for SetTemplate in Fig. 2. A kernel contract includes a mathematical model of each new type: the mathematical model of a Set is a finite mathematical set. Each operation's effect on its parameters is described in terms of their mathematical models via operation contracts, which state pre- and post-conditions in requires and ensures clauses, respectively (where # means "old"). An enhancement contract specifying an operation that computes Set union and intersection is shown in Fig. 3.

```
contract UniteAndIntersect
  enhances SetTemplate

  procedure UniteAndIntersect
                    (updates s: Set,
                     updates t: Set)
    ensures
      s = #s union #t and
      t = #s intersection #t

end UniteAndIntersect
```

Fig. 3: UniteAndIntersect Contract

The main element of the realization Iterative1 is shown in Fig. 4. Since implementations of enhancements are written from a client's point of view, their verification can be conducted in a fully modular fashion, making use only of the contract(s) of the component(s) they live on top of. In this example, the implementation of UniteAndIntersect is verified using SetTemplate contract.

```
procedure UniteAndIntersect
                  (updates s: Set,
                   updates t: Set)
  variable tmp: Set
  loop
    maintains
      s union t = #s union #t  and
      (s intersection t) union tmp =
      (#s intersection #t) union #tmp
      and t intersection tmp = {}
    decreases |t|
    while not IsEmpty (t) do
      variable x: Item
      RemoveAny (t, x)
      if not IsMember (s, x) then
        Add (s, x)
      else
        Add (tmp, x)
      end if
     end loop
  t :=: tmp
end UniteAndIntersect
```

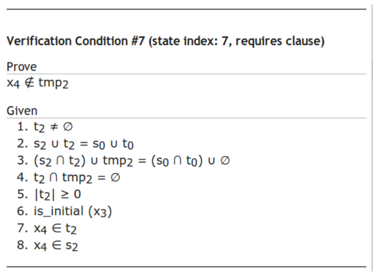
Fig. 4: UniteAndIntersect Code

As stated earlier, the RESOLVE language includes syntactic slots for certain types of code annotations, as is evident by the loop invariant (maintains keyword) and the termination metric (decreases). The :=: operator swaps the values of two variables.

2.2 VC Generation

We generate VCs automatically in accordance with the RESOLVE proof system defined formally in [10] and informally in [11]. This method begins by constructing a "symbolic tracing table" [11], which indicates, in terms of the program's abstract state, the assumptions and proof obligations at each observable state in the client program. Roughly speaking, assumptions are obtained from the `ensures` clauses of operations invoked in foregoing lines of code, whereas proof obligations are imposed by `requires` clauses of operations about to be invoked. Verification conditions are generated purely syntactically, and simplified by the VC generator only as described below; everything else required to prove a VC is carried out by a back-end automated proof engine.

The example VC shown in Fig. 5 corresponds to the point in `Iterative1` immediately prior to the line `Add(tmp, x)`. The "Givens" all represent assumptions at this point in the code, for example due to postconditions of preceding code, or based on the particular branch of the `if` we are in. For example, the 8th given, $x_4 \in s_2$, arises from the negation of the `if` condition, since we are in the `else` branch. Note that program variable values appear in VCs as subscripted mathematical variables, the subscripts being necessary to denote different values the variable may hold at different program locations. We do simple substitution of equals-for-equals prior to displaying VCs or sending them to a prover, which markedly reduces the number of extraneous subscripted variables.



Verification Condition #7 (state index: 7, requires clause)

Prove

$x_4 \notin tmp_2$

Given

1. $t_2 \neq \emptyset$
2. $s_2 \cup t_2 = s_0 \cup t_0$
3. $(s_2 \cap t_2) \cup tmp_2 = (s_0 \cap t_0) \cup \emptyset$
4. $t_2 \cap tmp_2 = \emptyset$
5. $|t_2| \geq 0$
6. is_initial (x_3)
7. $x_4 \in t_2$
8. $x_4 \in s_2$

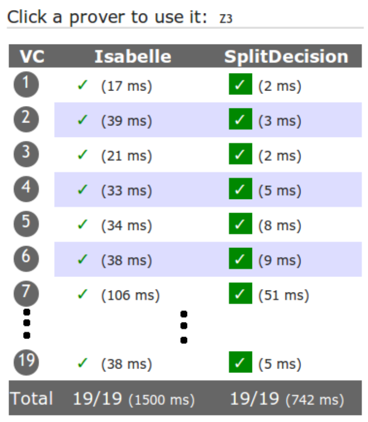
Fig. 5: Example of a typical VC

Since there is a loop invariant present in this code, its clauses are "Givens" for this VC because the state is inside the loop body; but these also appear elsewhere as proof obligations, *i.e.,* when considering the code points just before the loop and at the end of the loop body.

2.3 Provers

We attempt to prove or disprove VCs by appealing to two automated proving tools: Isabelle and SplitDecision. Isabelle [12] is a proof assistant that is capable of checking a proof that a user directs. While Isabelle is nominally an *interactive* proof assistant, it is possible to use it in an automated manner by augmenting each VC with a small, automatically generated Isabelle proof script to guide its actions. We do not use most of the theories available in Isabelle for the automatic proofs of VCs, but rather use Isabelle's proof engine along with fixed theories (integers, finite sets, finite strings) and mathematical units (*e.g.,* trees) developed for use in RESOLVE specifications [13]. By doing this, we can translate the VCs syntactically into input expected by other provers, *e.g.,* SplitDecision.

SplitDecision is a custom-built tool designed specifically for the simplification of RESOLVE VCs. SplitDecision leverages domain-specific mathematical knowledge along with general-purpose strategies for reducing logical formulae in order to simplify RESOLVE VCs in a truth-preserving manner. The simplifications SplitDecision performs are guided by specialized decision procedures tuned to handle the mathematical theories defined in RESOLVE and used in RESOLVE specifications. Thus far, SplitDecision performs simplifications that make use of simple arithmetic, and also implements a decision procedure for a substantial (decidable) fragment of string theory [14].



Click a prover to use it: Z3

VC	Isabelle	SplitDecision
1	✓ (17 ms)	✓ (2 ms)
2	✓ (39 ms)	✓ (3 ms)
3	✓ (21 ms)	✓ (2 ms)
4	✓ (33 ms)	✓ (5 ms)
5	✓ (34 ms)	✓ (8 ms)
6	✓ (38 ms)	✓ (9 ms)
7	✓ (106 ms)	✓ (51 ms)
⋮	⋮	
19	✓ (38 ms)	✓ (5 ms)
Total	19/19 (1500 ms)	19/19 (742 ms)

Fig. 6: `Iterative1` Results

The result of a proof attempt for the `Iterative1` code is shown in Fig. 6. In its current state, the tool suite appeals to two different proving tools, and indicates success or failure for each prover on each individual VC, with timing information. We can see that for this example, both Isabelle and SplitDecision prove the nineteen VCs fairly quickly, but there is no direct pattern of correspondence between their execution times on the individual VCs. It is not unusual, for code where every VC is not proved by both, to find that Isabelle is capable of proving some and SplitDecision others. For some statistics about VC proof success over hundreds of VCs from client code in category (3), see [15].

3 Current and Future Work

Currently, our VC generation supports only client usage of ADTs, not the construction of new ADTs. We are currently adding support for proof of correctness of data representations. This work will proceed in stages towards more and more sophisticated representation techniques. For example, the first version supports only abstraction functions rather than abstraction relations [16]. As we increase the sophistication of the tool support, we will also expand the mathematical theories used in the specification of components.

4 Acknowledgments

The authors are extremely grateful for the help of Jeremy Avigad, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Bill Ogden, Murali Sitaraman, and Hampton Smith. This material is based upon work supported by the National Science Foundation under Grants No. DMS-0701260 and CCF-0811737.

References

1. Hoare, C.A.R.: The verifying compiler: A grand challenge for computing research. J. ACM **50**(1) (2003) 63–69
2. Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental benchmarks for software verification tools and techniques. In: Proceedings of VSTTE 2008 (Verified Software: Theories, Tools, and Experiments), Springer-Verlag (2008) 84–98
3. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
4. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 4111 of Lecture Notes in Computer Science., Springer (2005) 364–387
5. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. PLDI **43**(6) (2008) 349–361
6. Zee, K., Kuncak, V., Rinard, M.C.: An integrated proof language for imperative programs. In: PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2009) 338–351
7. Weide, B.W., Heym, W.D.: Specification and verification with references. In: Proceedings SAVCBS. (2001) 50–59
8. Sitaraman, M., Weide, B.: Component-based software using RESOLVE. SIGSOFT Softw. Eng. Notes **19**(4) (1994) 21–63
9. Harms, D., Weide, B.: Copying and swapping: Influences on the design of reusable software components. IEEE Transactions on Software Engineering **17**(5) (May 1991) 424–435
10. Heym, W.D.: Computer Program Verification: Improvements for Human Reasoning. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH (December 1995)
11. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S.M., Hollingsworth, J.E.: Reasoning about software-component behavior. In: ICSR-6: Proceedings of the 6th International Conference on Software Reuse, Springer-Verlag (2000) 266–283
12. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
13. Heym, W.D., Long, T.J., Ogden, W.F., Weide, B.W.: Mathematical foundations and notation of RESOLVE. Technical Report OSU-CISRC-8/94-TR45, The Ohio State University (August 1994)
14. Friedman, H.: Some algorithms for strings with applications to program verification. Technical Report OSU-CISRC-8/09-TR42, Department of Computer Science, Ohio State University (August 2009)
15. Kirschenbaum, J., Adcock, B.M., Bronish, D., Smith, H., Harton, H.K., Sitaraman, M., Weide, B.W.: Verifying component-based software: Deep mathematics or simple bookkeeping? In Edwards, S.H., Kulczycki, G., eds.: ICSR. Volume 5791 of Lecture Notes in Computer Science., Springer (2009) 31–40
16. Sitaraman, M., Weide, B.W., Ogden, W.F.: On the practical need for abstraction relations to verify abstract data type representations. IEEE Trans. Softw. Eng. **23**(3) (1997) 157–170